THE CORRECTNESS PROOF OF A QUADRATIC-HASH ALGORITHM

by

A. N. Habermann

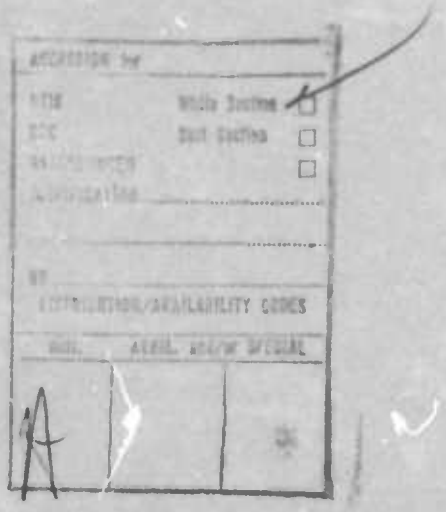
Carnegie-Mellon University

Pittsburgh, Pa. 15213

March 1975

DEPARTMENT
of
COMPUTER SCIENCE

Carnegie-Mellon University


ACCESSION for
NTIS
DDC
UNANNOUNCED
JUSTIFICATION
BY
DISTRIBUTION/AVAILABILITY CODES
DIST. AVAIL. and/or SPECIAL

THE CORRECTNESS PROOF OF A QUADRATIC-HASH ALGORITHM

by

A. N. Habermann

Carnegie-Mellon University

Pittsburgh, Pa. 15213

March 1975

Abstract. The statements of a program do not always provide sufficient information for proving its correctness. The correctness of the algorithm implemented by the program must often be proved with the pure mathematical techniques or exhaustive enumeration. An example program is presented for which the correctness proof of the program is trivial provided that the correctness of the underlying algorithm can be demonstrated. The program can be viewed as an abstraction of a quadratic hash algorithm. It is used at the end of the paper to encode the algorithm most efficiently.

This work was supported by the Defense Advanced Research Projects Agency of the Office of the Secretary of Defense under contract F44620-73-C-0074 and is monitored by the Air Force Office of Scientific Research.

1. Proving the correctness of a program by inductive assertion has become a generally accepted method. It was introduced by R. W. Floyd when he applied it to flowchart programs [1]. The method was considerably improved by C. A. R. Hoare. He expresses the meaning of program statements in terms of a set of axioms [2]. The truth of P{S}Q, where P and Q are predicates and S a program statement, can be derived from the axioms that apply to S. The significance of Hoare's improvement of Floyd's method lies in the fact that now programming language constructs are directly used in the correctness proof. One of the most practical axioms is the one that describes the **while** statement by $P\{S\}P \rightarrow P\{\textbf{while } B \textbf{ do } S\}\ P \wedge \text{not } B$. The fact that the condition B is false if (and when) the while statement terminates is often used in the subsequent part of the program. Its usefulness is nicely demonstrated in Dijkstra's program for computing the greatest common divisor of two integer numbers [3].

However, one must not get the impression that the inductive assertion method is complete in the sense that it shows that the algorithm, implemented by the program, is correct. The inductive assertions can do no more than show that the program is a true implementation of the algorithm. That is to say, the proof method can show that the program will execute the steps which the algorithm is supposed to take. If the algorithm is wrong, then the program will also be wrong. In addition to the correctness proof of the program, we also need to give a correctness proof of the algorithm. For instance, the correctness proof of the algorithm which Dijkstra uses to compute the gcd depends not only on the assertions between the program statements, but primarily on the mathematical facts that gcd(a,b) = gcd(b,a), gcd(a,a) = a and gcd(a,b) = gcd(max(a,b) - min(a,b), min(a,b)).

All correctness proofs have in common that long proofs are rarely convincing. This is true for the proof of a pure mathematical theorem as much as for the correctness proof of a program. Therefore, a convincing proof cannot be given unless the problem is decomposed in a coherent set of overseeable parts. This does not necessarily imply a top-down approach to writing programs, but it requires that various abstractions of the problem are considered separately. The abstractions must be manageable and they must be chosen such that the solution can easily be composed out of these. For small problems a decomposition based on the successive steps of the algorithm may be adequate, but Parnas shows that for large problems a decomposition based on the data structures and their operations is much more useful [4].

Sometimes the algorithms are simple and the programs rather complex because of pure size. An example of such a situation is the syntax scanner in the compiler for a programming language. The size of the code makes it necessary to split the program into smaller parts. Coming up with an exhaustive case analysis is one of the most difficult aspects of a correctness proof in this case. In this paper we consider the other extreme, where the correctness of the algorithm is not obvious, but the implementation is straightforward. The example presented in this paper serves two purposes. In the first place it shows that there is more to proving the correctness of a program than applying the inductive assertion method. Secondly, the resulting program has a practical aspect; it leads to a very efficient implementation of a quadratic hash technique.

2. Given the program

```
begin integer c,h,p;
  READ(p); READ(h);
 for c := 0 step 1 until (p-1)/2, (p-1)/2 step - 1 until 1 do
    begin h := (h+c)%p; PRINT(h) end
end
```

where h is a non-negative integer and p is a prime number such that p = -1 (mod 4). (E.g., the numbers p = 2↑k-1, where k is prime, satisfy these conditions.) The symbol / represents integer division and the symbol % represents the remainder function.

The question is to prove that this program prints all the numbers of the set S.0 = {0,1,2,...,p-1} exactly once, no matter which non- negative integer value is assigned to h in the statement READ(h). In other words, the program prints a permutation of the numbers 0,1,...,p-1, independent of the (permissible) initial value of h.

The relevant assertions are

A1: h is a number in the range [0..p-1]

A2: h is not equal to a number already printed

A3: all the printed numbers are different and in the range [0..p-1]

A4: the number of printed numbers is equal to p

With the assertion between { }, the program reads like this:

```
begin integer c,h,p;
  READ(p); {p is prime and p = -1 (mod 4)}
  READ(h); {h >=0}
  {A3}
 for c := 0 step 1 until (p-1)/2, (p-1)/2 step -1 until 1 do
    begin h := (h+c)%p; {A1}
            PRINT(h); {A2}
            {A3}
    end
  {A3,A4}
end
```

Assertion A3 must be an invariant of the for statement; it must be true after every iteration. The proof that all the assertions hold is simple for A1, A3 and A4, but difficult for A2. Assertion A3 is true before the iteration starts, because no numbers have been printed yet. The for clause shows that c>= 0 in every iteration and initially h >= 0. Thus, (h+c)%p is also greater than, or equal to, zero, so A1 is true. We can then show by induction that A1 is true for each iteration. Assuming that A2 is true, A3 holds at the end of an iteration. When the iteration terminates, A4 holds, because the **for** clause prescribes p iterations. The combination of A3 and A4 implies that all the numbers of S.0 = {0,1,...,p-1} have been printed exactly once. So, if assertion A2 is true, the program is trivial.

3. The proof that the program does not print a number in the range [0..p-1] twice cannot be derived from the program. But it can be proved by applying some number theory and group theory.

Let S.0 = {0,1,2,...,p-1}, S.1 = {1,2,...,p-1}, p is prime. Let the symbol ≡ represent equality modulo p. The zero-theorem says: if p is prime and a*b ≡ 0 for a,b ∈ S.0, then a ≡ 0 or b ≡ 0 (or both) [5]. The proof uses the fact that if a*b = m*p, then p must be a divisor of a or of b, since p is prime.

We split the iteration in two "parts". In the first part the iteration variable c = 0,1,...,(p-1)/2 and in the second part c = (p-1)/2, (p-3)/2,...,1. First we prove that two numbers h generated in the first part are not equal, then we prove the same for two numbers in the second part and finally we prove that a number h generated in the first part is not equal to any number generated in the second part.

a) Let h.i and h.j be two numbers generated in the first part. Let R.0 = {0,1,...,(p-1)/2} and let the initial value assigned to h in the statement READ(h) be H.

h.0 ≡ H, h.i ≡ h.(i-1)+i for i ≥ 1, so h.i ≡ H+i*(i+1)/2 for i ∈ R.0

If i,j∈R.0 and h.i ≡ h.j, then i(i+1) ≡ j(j+1), or (i-j)(i+j+1) ≡ 0. The zero theorem says that i ≡ j or i+j ≡ -1. Since i,j ∈ [0..(p-1)/2], i = j or i = j = (p-1)/2. Thus, if i,j ∈ R.0 and h.i ≡ h.j, then i = j. In other words, all the numbers h generated in the first part are different.

b) Let h.i and h.j be two numbers generated in the second part, where i,j ∈ R.1 = {1,2,...,(p-1)/2}. At the end of the first part, h has the value H + (p+1)*(p-1)/8. A number h.i in the second part is equal to

$h.i \equiv H + (p-1)*(p+1)/8 + (p-1)/2 + (p-3)/2 + \ldots + (p-(2i-1))/2 \equiv H + (p-1)*(p+1)/8 + i*(p-i)/2$

If $i,j \in R.1$ and $h.i \equiv h.j$, then $i(p-i) \equiv j(p-j)$ or $(i-j)(i+j) \equiv 0$. The zero theorem says that $i \equiv j$ or $i \equiv -j$. The latter equation has no solution for i and j both in R.1. The equality $i \equiv j$ implies for the given interval $i=j$. Thus, if $h.i \equiv h.j$ in the second part, then $i=j$ In other words, all the numbers generated in the second part are different.

c) Let h.i be a number generated in the first part and h.j a number generated in the second half. If $h.i \equiv h.j$, then

$i*(i+1)/2 \equiv (p-1)*(p+1)/8 + j*(p-j)/2$

$4i\uparrow 2 + 4i \equiv (p\uparrow 2-1) + 4pj - 4j\uparrow 2$

$(2i+1)\uparrow 2 + (2j)\uparrow 2 \equiv 0$

We must prove that the sum of two squares cannot be equal to zero modulo p. This is not self-evident. In fact, it is for many prime numbers possible that $x\uparrow 2+y\uparrow 2 \equiv 0$. E.g. for $p = 13$, $(x,y) = (5,1)$ or $(3,2)$ or $(11,3)$, etc. and for $p = 17$, $(x,y) = (1,13)$ or $(2,9)$ or $(3,5)$, etc. In the next section we prove that for $p = -1 \pmod 4$ the sum of two squares is not equivalent to zero.

4. It is well-known that the set $S.1 = \{1,2,\ldots,p-1\}$ with the operation "multiplication modulo p" is a (commutative) group if p is a prime number. (The operation is clearly associative and commutative. There remains to be shown that there is for every pair $a,b \in S.1$ an element $x \in S.1$ such that $a*x \equiv b$. Assume that the coset $V = aS.1$ has two equal elements: $a*c \equiv a*d$, where $a,c,d \in S.1$. If so, $a*(c-d) \equiv 0$. Since not $a \equiv 0$, $c \equiv d$ by the zero theorem. Thus, V has p-1 different numbers. Since all these

numbers are in the range [1..p-1], V = S.1. This means there must be an element a*y ∈ V such that a*x = b.)

Next we show that there is a pair x,y ∈ S.1 such that x↑2+y↑2 = 0 is true if and only if there is an element t ∈ S.1 such that t↑2 = -1.

Proof. Let x,y ∈ S.1 be a pair for which x↑2+y↑2 = 0 is true. There exists a t ∈ S.1 such that tx = y. Substitution yields x↑2(1+t↑2) = 0. The zero theorem says x↑2 = 0 or t↑2 = -1. The former is not true, because x ∈ S.1, so the latter follows.
Let t ∈ S.1 be such that t↑2 = -1. This implies t↑2+1 = 0. Thus, by substitution x = t and y = 1 we find a pair for which the sum of the squares is equal to zero modulo p.

Finally, we show that, if there is an element t ∈ S.1 such that t↑2 = -1, then p = 1 (mod 4).

Proof. The set G = {1,p-1,t,p-t} is a subgroup of S.1. This is immediately clear if we write out a multiplication table for G. It is well-known that two cosets V = aG and W = bG, where a,b ∈ S.1, coincide or have no elements in common [5]. (Assume V and W have an element c in common, so c = a*g = b*h, where g,h ∈ G. If k is the inverse of h, then k ∈ G, because G is a group. Then a = b*g*k is an element in G, so aG = b*g*kG = bG, so V and W coincide.) Therefore, the collection of all the cosets of G form a complete partioning of S.1. Each coset has the same number of elements as G. Hence, if subgroup G has k distinct cosets, k*n(G) = n(S.1), where n(Z) represents the number of elements of set Z. Since n(G) = 4 and n(S.1) = p-1, we have 4k = p-1, so p = 1 (mod 4).

Corollary. If we choose p prime and p = -1 (mod 4) then not p=1 (mod 4). It follows from the theorem that in this case S.1 does not have an element t such that

$t\uparrow 2 \equiv -1$, because $p \equiv 1 \pmod 4$ is not true. This implies that there is no pair $x,y \in S.1$ such that $x\uparrow 2+y\uparrow 2 \equiv 0$.

We appiy this result to part c) at the end of Section 3. For the given prime number $p \equiv -1 \pmod 4$, there is no pair of numbers $x,y \in S.1$ such that $x\uparrow 2+y\uparrow 2 \equiv 0$, so

$$(2i+1)\uparrow 2 + (2j)\uparrow 2 \equiv 0, \quad \text{iff } 2i+1 \equiv 2j \equiv 0$$

This is not true, because $2j \in S.1$. Thus, all the numbers generated in the first part of the iteration are different from all the numbers in the second part. This completes the proof of assertion A2 and also the whole correctness proof.

5. This correctness proof is a clear example of the case where the correctness of the algorithm is much harder to prove than that of the implementation. The program is at the same time an example of a useful abstraction which can be used, after some minor transformations, in other programs. The remainder of this paper shows how the program can be used to obtain an efficient quadratic-hash algorithm.

It is hard to specify exactly what may be called a "minor transformation" of a program whose correctness has been proved. The point is that we would like to transform the program into a usable form without having to redo the correctness proof. A minor transformation of the given program would be the replacement of the PRINT-statement by the assignment A[h] := 1, provided that array A is properly declared and initialized. The reader can imagine how that is done. Another minor transformation would be the replacement of the **for** statement by a **repeat** and a **while** statement, while adding the proper assignments to the iteration variable c. This transformation results in

```
begin integer c,h,p; READ(p); READ(h);
  c := 0;
  repeat h := (h+c)%p; PRINT(h); c := c+1 until c=(p+1)/2;
  while c ≥ 0 do c:=c-1; h:=(h+c)%p; PRINT(h) od
end
```

A slightly better version is obtained by using a more general form of **repeat** statement with the syntax

repeat S1* **until** BE **do** S2* **od**

where S* is a sequence of zero or more statements and BE a Boolean expression. The repeat statement is equivalent to the following sequence of ALGOL 60 statements.

L1: S1*; **if** BE **then goto** L2; S2*; **goto** L1; L2:

The generalized **repeat** statement first executes the statement sequence S1* and subsequently, as long as BE is true, it executes the sequence S2*; S1*. Note that the test precedes the repeated statements (as in a **while** statement) if S1* is empty; the test is at the end of the repetition if S2* is empty.

[I am sorry to hurt D.E. Knuth's feelings by using the closing delimiter "**od**" [8]. I do not share his misgivings towards the opening delimiter **do**. In the sequence **do begin** I read **begin** as an opening bracket, but not as a verb in the English language. (Do all native American programmers read **begin** as a verb? I wonder.) I started using the g*neralized **repeat** statement 2 1/2 years ago when the discussion was going on about **exit** and **leave** statements. The fact that I have used it ever since shows its usefulness to me. See also the appendix.]

The version in which the generalized **repeat** statement is used reads

```
begin integer c,h,p; READ(p); READ(h);
 c := 0;
 repeat h := (h+c)%p; PRINT(h) until c = (p-1)/2 do c := c+1 od;
 while c > 0 do h:=(h+c)%p; PRINT(h); c:=c-1 od
end
```

In this version h is printed for c = 0,1,...,(p-1)/2 by the **repeat** statement and for c = (p-1)/2,...,1 by the **while** statement. Note that at the end of the **repeat** statement c is not unnecessarily incremented to (p+1)/2.

6. A quadratic hash algorithm starts off with computing some number hash.0 = H(name) as function of the machine representation of "name". As long as no match or free slot is found in the symbol table ST, the algorithm computes successive numbers by the rule

$$hash.i = hash.0 + a*i\uparrow 2 + b*i + c \pmod p$$

where p is the length of the symbol table.

The coefficients b and c have no impact on the probability Pr(k) that a match or an empty slot is found in precisely k = i probes. This has been argued by C. E. Radke [6] and J. R. Bell [7]. Thus, the coefficients b and c can be arbitrarily chosen. We choose c = 0. Coefficient a must be unequal to zero, otherwise the hash algorithm degenerates to a linear hash.

A well-known problem of the rule above is that it generates only (p+1)/2 different values for all i $\in$ S.0 = {0,1,..,p-1}. If we choose a = 1, b = 0, then all possible

squares modulo p are generated by the numbers $i \in \{0,1,...,(p-1)/2\}$, because i and (p-i) produce the same square. (All the repetitions are generated by the "upper half" $\{(p+1)/2,...,p-1\}$.) The missing numbers are found in the following way. The set $Q = \{x \uparrow 2 \pmod p \mid x \in S.1\}$ is a subgroup of S.1. Let $t \in S.1-Q$. The coset tQ has no elements in common with Q (because, if it had, tQ and Q would coincide which implies $t \in Q$). The coset tQ has the same number of distinct elements as the subgroup Q, (p-1)/2. Thus, $tQ \cup Q = S.1$, or $tQ = S.1-Q = S.0-Q \cup \{0\}$. This means that tQ contains all the missing numbers.

If we choose $p = -1 \pmod 4$, we know that -1 is not a square. In that case the remaining numbers can be found by choosing coefficient a = -1. A program that implements the quadratic hash for a = (1,-1) and b = 0, is

```
integer procedure QH(name); value name; integer name;
comment name is the internal representation of an identifier. The procedure returns
  -1 if the Symbol Table is full and the name is not in the table, otherwise it returns
   the index of the ST-entry that holds the name;
begin integer hash,index,c;
 hash := H(name); comment the initial hash function remains unspecified here;
 c := 0; comment the constant p and the symbol ST are global objects;
 repeat index := (hash+c*c)%p
 until ST[index] = name or ST[index] = 0 or  c = (p-1)/2 do c := c+1 od;
 if ST[index] ≠ name and ST[index] ≠ 0 then
   c := 1; index := hash;
   repeat index := (hash-c*c)%p; if index < 0 then index := index+p fi
   until ST[index] = 0 or ST[index] = name or c = (p-1)/2 do c := c+1 od
fi;
```

```
  comment we follow the ALGOL 68 convention of terminating an if statement by fi;
  if ST[index] = 0 then ST[index] := name fi;
  QH := if ST[index] = name then index else -1 fi
end
```

One might argue that the new value of index can be computed from the old value of index by adding (or subtracting) 2c+1. This is true, but the sum should be written as c+c+1, otherwise the computation is even longer than c*c! The variable index cannot be deleted, because the original value of hash is needed before the second iteration begins.

We find significant improvements of the quadratic hash program if we observe the similarity between this program and the one whose correctness we proved. Noticing the similarity allows us to delete the variable index, the conditional test in the second iteration, and it allows us to replace the product c*c by the single term c. The resulting program for QH is

```
integer procedure QH(name); value name; integer name;
begin integer hash,c;
  hash := H(name); c := 0;
  repeat hash := (hash+c)%p
  until ST[hash] = 0 or ST[hash] = name or c = (p-1)/2 do c := c+1 od;
  while ST[hash] ≠ 0 and ST[hash] ≠ name and c > 0 do
    begin hash := (hash+c)%p; c := c-1 end;
  if ST[hash] = 0 then ST[hash] := name fi;
  QH := if ST[hash] = name then hash else -1 fi
end
```

Conclusion

The correctness proof cannot always be derived from the program structure. It is often necessary to prove the correctness of an algorithm by pure mathematical methods or by exhaustive enumeration. Once the correctness of the algorithm has been established, the correctness proof of the program amounts to proving that the program is a true implementation of the algorithm.

A correctness proof becomes intractable if the problem is not split into a coherent set of subproblems which are of smaller complexity than the total problem. The subproblems are constructed by modeling a small aspect of the total problem and by abstracting from the total problem everything that is irrelevant to this model. The art of finding the right abstractions is of greater importance than the art of finding the right assertions. If a programmer masters the former, he will have no problem with the latter, but the opposite is not true.

There remains the problem of composition. The programmer must make sure that the resulting program does not violate the conditions that allowed him to prove the correctness of the abstractions. In most cases this amounts to showing that some trivial transformations of the correct programs again result in correct programs. It is as yet not clear which transformations we may consider as trivial in the sense that the correctness of the transformed program follows from the correctness of the original program.

Appendix

The generalized **repeat** statement used in Section 6 is a special case of the iterative statement. The various forms of the iterative statement are

```
{ for <var> in {-}<range> } repeat S1* until BE { do S* od }
{ for <var> in {-}<range> } while BE do S* od
for <var> in {-}<range> do S* od
```

The { } pair indicates optional parts. S* means a sequence of zero or more statements. The iterative statement can be optionally preceded by the prefix { with <decl>* }. This optional prefix allows the programmer to declare data local to the iterative statement. A range usually has the form [<indexexpression> . . <indexexpression>]. It may also be an array name, an expression list or a range type name. A range type is defined by an enumeration of the constants of that type (e.g. **range** day = Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday **end**). The optional minus sign in front of the range indicates whether the iterative variable should step through the range in reverse order.

The iterative variable in the for-clause must be a declared variable of the same type as the range elements. No matter whether the iterative variable is changed in the sequence of statements controlled by the for-clause, before the next iteration commences, the next value in the range is assigned to the iterative variable. A range defined as a pair of index expressions is considered to be empty if the expression left of the separator ".." is greater than the expressions to the right of the separator. The for-statement is in that case quivalent to an empty statement.

In addition to these iterative statements we found it useful to have two iterative Boolean expressions which correspond to predicates prefixed by a single logical quantor. The syntax of these expressions is

some <var> in {-} <range> **sat** BE

all <var> in {-} <range> **sat** BE

The delimiter **sat** is an abbreviation for "satisfy" or "satisfies". The first expression corresponds to a predicate prefixed by "there exists", the second to a predicate prefixed by "for all". The expressions are not independent; the one can be expressed by the negation of the other if the Boolean expression BE is replaced by not BE. However, it is convenient to have both and the one is not more difficult to implement than the other. The iterative variable is a declared variable of the same type as the range elements. If the variable steps through the entire range, the final value is undefined. Otherwise, it has the first value which made the first expression true or the second expression false.

Some examples of applications. Let a square matrix A[0..p,0..p] be given. The following expression returns true if every row has at least one element equal to zero.

all x in [0..p] **sat** (**some** y in [0..p] **sat** A[x,y] = 0);

Application is to the quadratic hash algorithm leads to the following program.

```
integer procedure QH(name); value name; integer name;
begin integer c,hash; hash := H(name);
QH :=
 if some c in [0..(p-1)/2] sat ST[hash := (hash+c)%p] = 0 or ST[hash] = name
  then hash
 else
  if some c in - [1..(p-1)/2] sat  ST[hash := (hash+c)%p] = 0 or ST[hash] = name
   then hash else -1 fi fi;
 if ST[hash] = 0 then ST[hash] := name fi
end
```

The last example is a procedure SPACE which searches through the global bit array B[0..top-1] for k contiguous bits equal to one. If found, it returns the index of the first bit of this area ; if not found, the procedure returns -1.

```
integer procedure SPACE(k); value k; integer k;
begin integer x,y,z; x := 0; z := k;
while z<=top do
 if some y in -[x..z-1] sat B[y] = 0
 then x:=z; z:=y+k+1 else RETURN(z-k) fi od;
RETURN(-1)
end
```

The program is slightly faster if y steps through the range in reverse order, because in that case y points the rightmost zero in the inspected area. Variable y steps through the range in reverse order if the range expression is preceded by a minus sign.

References

[1] Floyd, R. W., "Assigning Meaning to Programs," **Proc. Amer. Math. Soc. Symposia in Applied Mathematics,** 19 (1967).

[2] Hoare, C. A. R., "An Axiomatic Basis for Computer Programming," **CACM** 12, 10 (October 1969).

[3] Dykstra, E. W., A Short Introduction to the Art of Programming, Technological University Eindhoven (1971).

[4] Parnas, D. L., "On the Criteria to be Used in Decomposing System Modules," **CACM** 15, 12 (December 1972).

[5] Loonstra, F., **Introduction to Algebra,** P. Noordhoff N.V., Groningen.

[6] Radke, C. E., "The Use of Quadratic Residue Research," **CACM** 13, 2 (February 1970).

[7] Bell, J. R., "The Quadratic Quotient Method: A Hash Code Eliminating Secondary Clustering," **CACM** 13,2 (February 1970).

[8] Knuth, D. E., "Structured Programming with Goto Statements," **Computing Surveys** 6, 4 (December 1974).

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: AFOSR-TR-75-1086	2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE (and Subtitle): THE CORRECTNESS PROOF OF A QUADRATIC-HASH ALGORITHM	5. TYPE OF REPORT & PERIOD COVERED: Interim
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s): A. N. Habermann	8. CONTRACT OR GRANT NUMBER(s): F44620-73-C-0074, ARPA Order-2466
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Carnegie-Mellon University, Dept. of Computer Science, Pittsburgh, PA 15213	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 61101D, AO 2466
11. CONTROLLING OFFICE NAME AND ADDRESS: Defense Advanced Research Projects Agency, 1400 Wilson Blvd, Arlington, VA 22209	12. REPORT DATE: March 1975
	13. NUMBER OF PAGES: 18
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): Air Force Office of Scientific Research (NM), 1400 Wilson Blvd, Arlington, VA 22209	15. SECURITY CLASS. (of this report): UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The statements of a program do not always provide sufficient information for proving its correctness. The correctness of the algorithm implemented by the program must often be proved with the pure mathematical techniques or exhaustive enumeration. An example program is presented for which the correctness proof of the program is trivial provided that the correctness of the underlying algorithm can be demonstrated. The program can be viewed as an abstraction of a quadratic hash algorithm. It is used at the end of the paper to encode the algorithm most efficiently.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

403 087